# Flash Aware DataBase Management System

**A Dissertation submitted in partial fulfillment of the requirements for the award of**

**Master of Technology**

**in**

**Computer Science & Engineering**

***By***

**Prerana Tiwari**

**CS10m04**


**Department of Computer Sciences & Engineering**

**Indian Institute of Technology Hyderabad**

**June-2012**

## Declaration

I declare that this written submission represents my ideas in my own words, and where ideas or words of others have been included, I have adequately cited and referenced the original sources. I also declare that I have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified any idea/data/fact/source in my submission. I understand that any violation of the above will be a cause for disciplinary action by the Institute and can also evoke penal action from the sources that have thus not been properly cited, or from whom proper permission has not been taken when needed.

Tiwari

Signature

CS10M04

RollNo

Poorvaja Tiwari

Name

Approval Sheet

This thesis entitled "Flash Aware Database Management System" by Prerana Tiwari is approved for the degree of Master of Technology from IIT Hyderabad.

Ashmdib Dutta

-Name and affiliation-
Examiner

-Name and affiliation-
Examiner

-Name and affiliation-
Adviser

-Name and affiliation-
Chairman

**Acknowledgement**

I am deepely indebted to my adviser Dr. Ravindra Guruvanavar for his constant guidance and support right from my problem selection to report documentation. He patiently listened to all the problems I faced during the course of this work and appropriately guided me with his full support. I thank him for patiently clearing all my novice doubts.

I thank all the faculty members of the Department of Computer Science and engineering for sharing their views and giving valuable suggetions during the discussion of my work in department seminars.

I thank all my classmates for their friendly support who made the stay at this institute enjoyable, we shared joy and knowledge. I thank all my friends at IIT Hyderabad for the same.

I thank our director Prof. U.B.Desai for his friendly administrative support in getting all our requirements done as quickly as possible.

Finally, I thank all my family members for their constant love and support. iv

**Abstract**

Flash Memory is valued in many application as a storage media due to its fast access speed, low power, nonvolatile characteristics.Our survey report will contain characteristics of flash disk, architechture of flash disk, various indexing structure for magnetic disk and flash disk, storage techniques for hard disk and magnetic disk and query processing techniques for flash disk. we have explored detail survey of storage, indexing and query processing techniques developed to make database system flash aware. We have implemented most of the techniques in a database system prototype named Mubase developed at IITH. We present some experimental results on TPC-H dataset demonstrating the benefits due to the flash aware storage query processing techniques. We have implemented FD - Tree index structure for flash disk on prototyped named Mubase.

# Contents

# Chapter 1

# Introduction

Flash memory is an electronic device based on semiconductor technology. It is a type of nonvalatile, electrically erasable programmable read only memory (EEPROM) that has been used in consumer devices like thumb drives, PDA's, cameras, and mobile phones. Flash disk also known as solid state disk (SSD). It falls between RAM and hard disk in terms of acquisition cost, transfer bandwidth, spatial density,cooling cost.[Grae07] . It is packaged in hard disk form and uses the same standard host interface used in traditional hard disk [leve08].

Flash disk is pure electronic device, it does not have mechanically moving parts like the read/ write head that increases latency in magnetic disk, because of this property of flash disk random data access is very fast compare to magnetic disk. Moreover, flash disk is more compact in size, and has better shock resistance than magnetic hard disk.Write operation in flash memory is quite expensive because of intrinsic property of flash memory. Pages in flash disks are units of read and write operations. A page in a flash disk can have either of two states - erased or non-erased (dirty) [Myer 07]. Data can be written to a flash disk page only if when page is in an erased state. Flash memory is organised as set of blocks. A block is an erase unit of flash disk. An erase unit can contain as many as 256 or more pages. When we want to update any page of flash memory, that block which contains particular page should be in erased state. For erasing that block all the pages of that block will be written into main memory, than after updating pages all the pages will be written back to the disk. Erase is very costly as compared to read and write operations (it takes approximately 1.5ms to erase a block). Further the number of erase cycles per

erase block is limited, typically ranging from 10, 000 to 1, 000, 000. After the cycle limit has been exceeded, the erase block burns out, and it becomes useless.

For increasing the write performance of magnetic disk based on B+ tree some index structures have been proposed. B+ tree is a balanced binary search tree which gives very less lookup cost. For example, LSM-tree, stepped merge tree, where random writes are limited to fixed area and large amount of data is sequentially written to the hard disk.

In this report we have explored detail survey of storage, indexing, and query processing techniques to make the database system flash aware. We have implemented most of these techniques in a database prototpe named mubase developed at IITH. We present experimental results on TPC-H data set demostrating the benefits due to flash aware storage, indexing, queryprocessing techniques.

# Chapter 2

# Flash Memory

Flash memory is a type of nonvolatile memory i.e. it does not need power to maintain information in it. There are two types of flash memory:-

- NOR-Type Flash Memory
- NAND-Type Flash Memory

NOR type flash memory is bit adrresable and has a fully memory mapped random access interface with dedicated address and data lines allowing the cells to be programmed bit-by-bit. It is used for storing codes since it is directly addressable by processors. NAND-type flash memory has no dedicated address lines and is controlled by sending commands and addresses through an indirect IO-like interface. It is accessed on a page bassis (typically 512 bytes to 4 Kbytes) and provide higher cell densities.Therefore, NAND- type flash memory behaves as a block based device similar to magnetic drives [Lee 07]. The NAND type is primarily used for removable flash cards, USB thumb drives, and internal data storage in portable devices.

NAND based flash disk are cheaper and in common use today and come in two varieties, namely SLC (single level cell) and MLC(multi level cell)[Leve 08].SLC can store single bit per memory cell whereas MLC can store multiple bits per cell.Rather than simply being on or off each transistor in MLC NAND is able to enter one of four states allowing them to encode data to achieve a storage density of two bits per memory cell, which effectively doubles the capacity of NAND flash memory. SLC flash disk costs twice as much as MLC flash. However, it is much more reliable and has much better endurance (write/erase cycles). It is also much faster (in terms of read/write speed) than MLC flash [Mosh 08]. Therefore, SLC flash disk is suitable

for enterprise systems. Throughout the rest of this thesis, we will use solid state disk (SSD) or simply flash disk to refer to NAND-based solid state disk (NAND-based flash disk).

## 2.1 Properties Of Flash Memory

1. **No Mechanical Latency:-**The solid state disks are purely electronic based device, thus they do not have any moving parts, so that there is no rotational and seek latencies in the flash disk.On the other hand magnetic disks have mechanically moving parts like read write head. Time necessaryto move the disk arm to the desired cylinder called seek time and the time necessary to move to the desired sector to rotate the disk head called rotational latecy. Because of these rotational and seek latencies random access time of magnetic disk are much more high.On the average rotational latecies are 4.2 ms for 7200RPM, 3 ms for 10,000 RPM and 2ms for 15,000 RPM. Seek time for these disks ranges from 3ms to 10 ms. Therefore performance of SSD is better than magnetic disk.

2. **No In-Place Updates:-** Flash memory consists of set of blocks. Size of block ranging from 128KB to 256 KB and it consists of set of pages. Bits in flash memory is organized in to pages of size ranging from 512B to 2KB. Data read and write operation in flash disks are done at the granularity of flash pages.

   Flash pages have only two possible states: erased and non erased. In the erased state all bits are considerd to be 1. Only in a erased state data can be written. Writes to a page can only clear bits (reset them to 0) and can not set a bit back from 0 to 1. Therefore page can not be updated in-place or overwritten after it is written once unless it is somehow brought back in to an erased state.

   When we want to update any page first page containing that block should be in erased state.This erase operation will set all bits in the block to 1. Erase operation is much more costly than read and write operation.

3. **Assymetric speed of reads and writes:-** Although random read operation

on flash disks are significantly faster than on magnetic disks, write operation on flash disks, especially small random writes, exibhit relatively poor performance. The read speed of flash memory is typically at least twice as fast as the write speed. This is because it takes longer time to inject charge into the memory cell than reading its status.[Lee07,North 08]

4. **Limited Number Of Writes:-**A block in flash disk has a limit to the number of erase cycles it can sustain before it becomes unusable. MLC flash supports up to 100, 000 erases per block whereas an SLC flash is typically rated at 1, 000, 000 to 5, 000, 000 program or erase cycles per block. Flash memory devices use a technique known as wear- leveling in order to ensure that cells are stressed uniformly so that hot cells do not cause premature device failure. Flash devices keep a pool of spare blocks for bad-block remapping. Most flash devices also have the capability to estimate their own remaining lifetime so that a system can anticipate failure ahead of time and take necessary precaution measures. [Leve 08]

## 2.2 Flash Disk Architecture

A Flash based SSD is constructed as an array of flash packages. A Flash packages consists of one or more dies or chips. In the following diagram 4 GB flash disk consistiing of 2 GB of dies, sharing an 8-bit serial I/O bus and a number of common control signals. The two dies have separate chip enable and ready/busy signals. Thus one of the die can accept commands and data while other is carrying out another operation. The package also supports interleaved operations between the two dies.

Each die whinthin a package contains 8192 blocks, organized among 4 planes of 2048 blocks.The dies can operate independently, each performing operations involving oone or two planes. Two planes command can be executed o either plane- pairs 0-1 or 2-3, but not across other Each block in turn consist of 64 4KB pages. In addition to data, each block includes 128 byte region to store metadata.

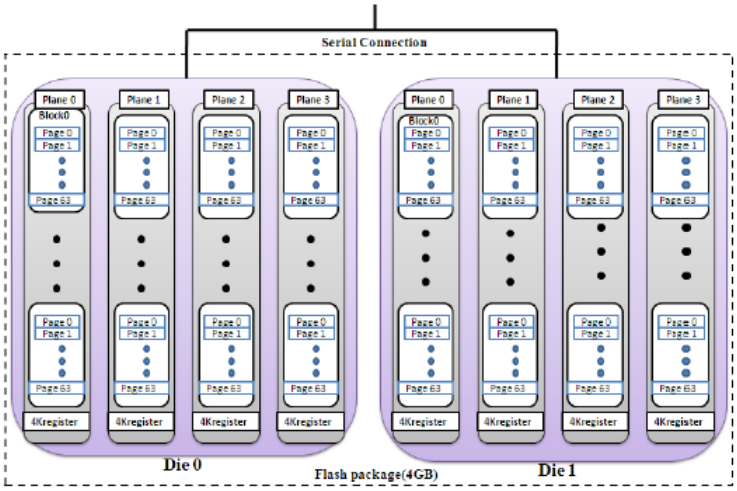



Figure 2.2: Samsung 4GB flash internals [Agra 08]

Flash packages receive commands and transmit data over a serial interface. During a read operation, data is first transformed from a place to associated on chip register. The data is than transffered to an off chip controller via a serial interface. The exact reverse process take place during write operation. Each of the chips have separate enable and ready/busy signals.This feature enables interleaving of operations among the different chips in a package thus providing a considerable speed up. An SSD can be connected to a host using one of different possible host interconnect like USB, FiberChannel, PCI Express or SATA. An internal buffer manager and a multiplexer are used to match the limited bandwidth of interface with the performance available from a array of flash packages.

## 2.3 Flash Translation Layer

Flash based SSDs emulate mechanical disks using a structure called logical block mapping technique. Static logical block mapping and dynamic block mapping. Physical to logical block mapping is performed for accessing data residing in the physical memory of the magnetic disk. In-place update is possible in magnetic disk, new data is written in a given LBA and same LBA is used to retrieve data. In place update is not possible in flash based disk. Every write operation in flash based SSD requires some form of mapping between logical block address and physical flash page allocation. There are two kinds of block mapping techniques

### 2.3.1 Static Block Mapping

In static block mapping, LBAs are directly mapped to the page IDs of the flash disk. Since no inplace update is possible in flash disk each update in a page requires erasing block that contains that page. Before erasing, that block is written in to the main memory than after erasing modified data is again written back to the disk. Given that an erase block consists of 64 pages, such strategy imposes a massive overhead. This technique can also lead to premature failure of blocks containing hot pages.[Myer 07]

### 2.3.2 Dynamic block mapping

In a Dynamic Block Mapping LBAs can be mapped to page IDs of flah disk dynamically using FTL(flash translation layer). Since no inplace update is allowed in flash disk. Updated page should be written to a free page and page ID is dynamically mapped to LBA. Such dynamic mapping of LBA to page ID is possible using FTL(flash translation layer). FTL is a translation layer between the file system and the flash disk that work in the cooperation with operating system in order to make flash disk appears like magnetic disk to the application. In reality write in place is not possible in flash but FTL manages data on flash so that it appears like a magnetic disk. During write operation data is written to available free page. The persistent mapping record entry for the LBA is than updateed with this new page's ID and the old flash page is marked as dirty or garbage. In case of crash recovery garbage collection is possible through FTL. Since each block has some fixed life time. Wear-leveling is performed in the blocks to use all the blocks evenly using FTL.

# Chapter 3

# Indexing Structure For Magnetic Disk

Indices are used to retrieve data from large database. B+ tree is more popular data structure used as a index structure to retrieve data.

## 3.1 B+ tree index structure

B+ tree is a balanced tree having a large fanout. All the records are stored into the leaf level. and internal nodes contain pointers to the next level. B+ tree node has n-1 key values and n pointers that will point to the next level. A pointer $P_i$ in leaf node points to the record with key value $K_i$. Pointer $P_{n+1}$ in leaf nodes is used for chaining the leaf nodes in order to facilitate range search. For inserting value in B+ tree we find out appropriate node location in B+ tree. If node does not have space to locate that entry node split operation is performed in that node.Node is devided into two left and right node. First element of the right node is inserted into the parent node. If parent node exceed its capacity parent node is splited.

In deletion if node size is less than n/2 than merge operation is performed. For inserting and deleting value from the B+ tree, tree nodes are fetched from hard disk to the main memory and than written back to the hard disk. Since B+ tree index structure is the most commonly used structure and gives fast read performance for search intensive applicatione.

## 3.2 Log Structured Merge Tree

Log structured merge tree also known as LSM - tree , proposed in [ONei 96] is an index structure that defers individual insert and/or update operations and migrates them later to disk in batches. It is mainly designed for applications where search queries are less frequent than insert/update operations.B+ tree index structure is not efficient for such application. A LSM - tree is composed of two or more tree - like component data structure.

### 3.2.1 The Two Component LSM - Tree Algorithm

In two component LSM - tree one component is memory resident called $C_0$ and other is disk resident called $C_1$. First new entry is written to the $C_0$ tree and when the capacity of $C_0$ exeeds the entries are migrated to the $C_1$ tree. A rolling merge operation is performed and entries from the $C_0$ tree is deleted and it is merged with the $C_1$ tree.

The rolling merge operation is a series of merge steps. Each merge step then reads a disk page sized leaf node of the $C_1$ and buffered it in a block, merges entry from the leaf node with entries taken from the leaf level of the $C_0$ tree, thus decreasing the size of the $C_0$ and creates new merged leaf tree in $C_1$.

New merged child tree is written to the entirely new location in the $C_1$ tree. and old child nodes of the $C_1$ tree that has involved in merging are kept for recovery in crash, and parent of that old child tree is now assigned to the new tree.This paent node of $C_1$ also buffered in memory, and frequently referenced page nodes in $C_1$ will remain in memory buffer, so that popular high level directory nodes of the $C_1$ can be counted on to be memory resident.

### 3.2.2 Deletion In LSM-Tree

In deletion we search that key value and record ID in $C_0$ tree, if it is found we delete it from that position, if it is not found in $C_0$ tree same entry is inserted to the appropriate position in the $C_0$ tree. While rolling merge process same entry will

be encountered than both old and new entry will be discarded.

## 3.3 Stepped-Merged Tree

It is an another variant of LSM -tree, that is used in database where writes are more frequent and large amount of data has to be stored for example datawarehouse and recording system where the value of index entry are randomly distributed. This indices are designed for write intensive indices to minimize the write overhead while steel providing a relatively efficient speed. Organizing the data on disk as the data arrives using standard techniques would result in more that one I/O which is very expensive when the data arrive rate is very high . A commonly work used in dataware houses is to collect the record and update the database only periodically. Instead of performing each write operation when data arrives we group the data and finally merge them into the disk. A stepped merge B+ tree effectively handle this situation by providing fast insertion/update operation in a reasonable time.The basic principle behind this tree is external merge sort.

### 3.3.1 Algorithm For Stepped Merge B+ - Tree

1. First collect the incoming data in to the main memory and call it current run.When memory is full call it previous run and store in a disk by constructing a b+ tree structure and start storing new incoming data in main memory in current run. The B+ tree is constructed in a bottom up since the data is sorted. Both the in memory run and the one just constructed is called Level 0 runs.

2. When at level i k runs are there merge them in to a single run and make first entry of the level i+1. After merging delete the old run.

3. When N level K-1 runs are accumulated, merge them, but instead of writing them in to a new run, insert the entries in to the root relation. The root relation is also organized using a B+- tree file organization.

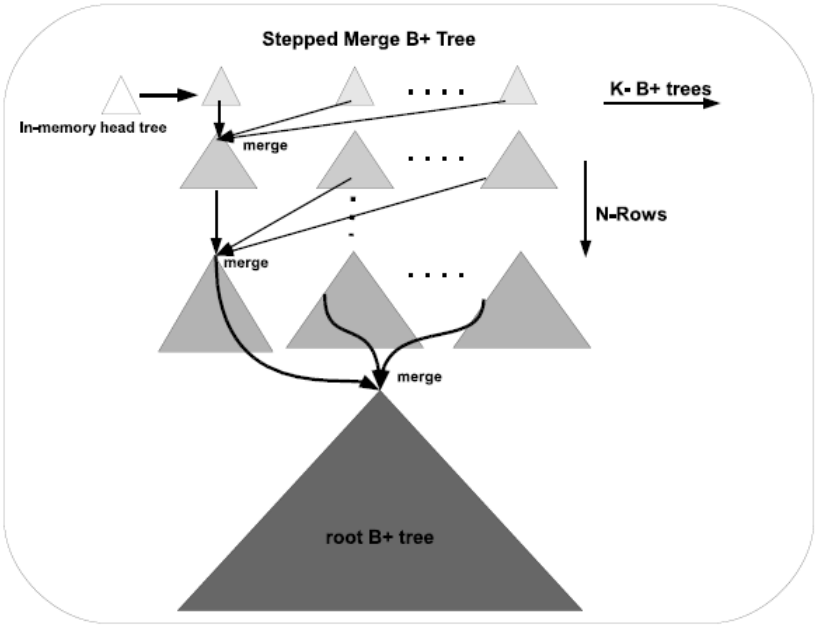



Figure 3.1: Write optimized $B^+$ tree index structure

Single lookup cost is (K*N+1)*($T_s$ + $T_t$) .
[jaga]

### 3.3.2 Run Index

A run index stores pointers to all the runs currently in the existence. including the run currently being constructed in main memory. When K runs are merged to get a single run at a higher level, pointers to the K runs are deleted from the run index, and replaced by a pointer to the single higher level run. And when a new run is created in memory, a pointer to it is added to the run index. All the trees together with the run index constitute a multi tree index.

We now consider some optimization. While creating the Kth run of Level i, instead of writing out to the disk and reading back from the disk for merging it can be directly merged with other level i runs. As a result recursively applying this optimization runs of several levels may get merged simultaneously.Applying the optimization to multiple levels, one in memory run of level 0 and k-1 runs of each levels 0...i on disk, will get merged to form a single run of level i+1 on disk. Than no level will have more than k-1 runs at a time in this optimization. Look up cost will be N(K-1) B+ tree and the last root B+ tree.

# Chapter 4

# Index Structure For Flash Disk

As we have discussed many index struture for magnetic disk like B+ tree. Index sturcture used for magnetic disk is not suitable for flash disk because of the intrinsic property "no in place write" of the flash disk. Write operation is very expensive in flash disk. So to reduce the cost of the write operation some index structures have been proposed .

## 4.1 $\mu$ Tree Index Structure

A B+ tree is implemented in flash disk withput its own FTL. An update to the leaf node leads to a chain of update on index nodes up to the root node because of the intrinsic property of flash disk i.e no in-pace update is allowed. They call this method of updating tree "wandering tree". Here updated node is written in a new free location and old location is marked as garbage collection.From parent of that node up to the root node is written in new free location. Since the parent node to the leaf node contain pointer to the old location , this pointer value need to be updated to the new location of of the child leaf node. The cost of this process is very high.

$\mu$ tree proposed in [kang 07] is a balanced tree indexing scheme for flash disks mainly degined to address the suboptimal wandering tree update method. $\mu$ tree is a balanced tree similar to B+ tree and is degined for both SLC and MLC nand flash memory to solve the inefficieency in wandering tree. $\mu$ tree arranges the layout so that updated nodes fit into a single flash memory page. We define the level of node in $\mu$ tree as the length of the path from leaves to the node plus one. The level of a

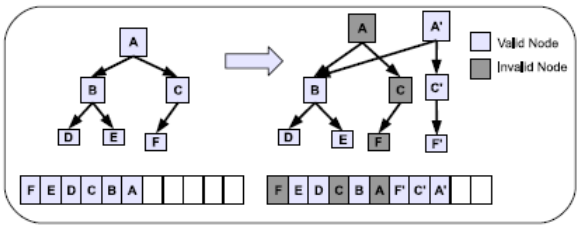



Figure 3.2: A wandering tree update example [Kang 07]

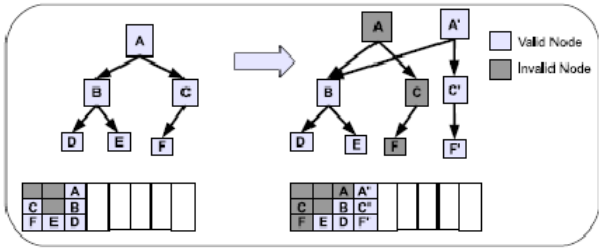



Figure 3.3: A $\mu$ -Tree update example [Kang 07]

leaf node is always one, height of the $\mu$ tree is defined as the level of the root.The size of each node in $\mu$ tree varies depending on the level of the node and the height of the $\mu$ tree. This is because $\mu$ tree should be able to store all the nodes in a path from the root to any leaf in to a single page. Suppose the height of $\mu$ tree as H, and the set of nodes in level L as $N_l$. For $\mu$ - tree such that H>2, a leaf node n $\in N_1$ always occupies the half of the page.As the level is increased, the node size is reduced by half. For an index node $m_L$, such that $m_L \in N_L$, 1<L< H, the size of the node is reduced by half compared to its children at the next level L-1. Only the root has the same size as its children nodes. When $\mu$ tree consists of consists of single level the entire flash page is used for root node.[kang07]

## 4.2 BFTL index structure

We present B-tree layer over flash memory to reduce the redundant data written to the flash memory because of hardware restriction of flash memory.There is a B-tree layer called BFTL over FTL that provides file systems to create and maintain B-tree index structure.[Wu07] BFTL consists of reservation buffer and node translation table. First keys and records are inserted to the BFTL then block wise requests are sent to the FTL from BFTL.Primarily records will be in reservation buffer.When

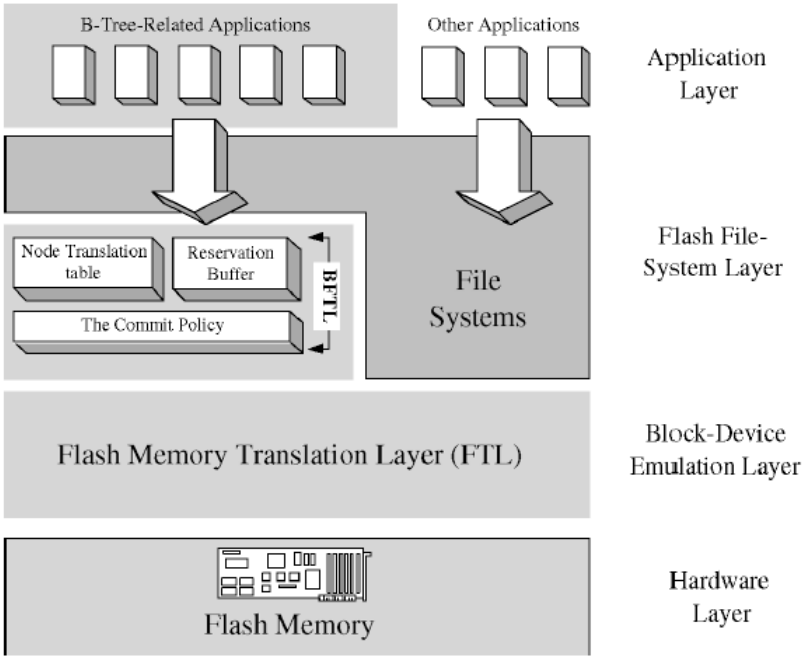



Fig. 2. Architecture of a system which adopts BFTL.

it reaches to its threshold entry it need to be flushed to the flash memory.To flush out dirty records in the reservation buffer, BFTL construct a corresponding "index units" for each dirty records, the records are written to allocated location.Since index units are very small they can be grouped in to few sectors to reduce the no of pages physically written.Sectors are the are logical items, which are provided by the block device emulation of FTL. We try to pack index units belonging to different B-tree node in a small number of sectors.Now the no of updates will be reduced, index units of one B-tree node could now exitst in different sectors. To construct the logical view of a B-tree node, relavant index unit is collected and parsed.An index units has several feilds data_ptr, parent_node, primary key, left_ptr, right_ptr, an identifier, and an op_feild. identifier of an index unit denotes to which B-tree node belongs.The op_feild will show operation could be done insertion, deletion,insertion or an update.In addition, time stamps are added for each batch flushing of index units to prevent BFTL from using stale index units. Index units could be scattered over flash memory. The logical view of B-tree will be constructed through the help of BFTL.

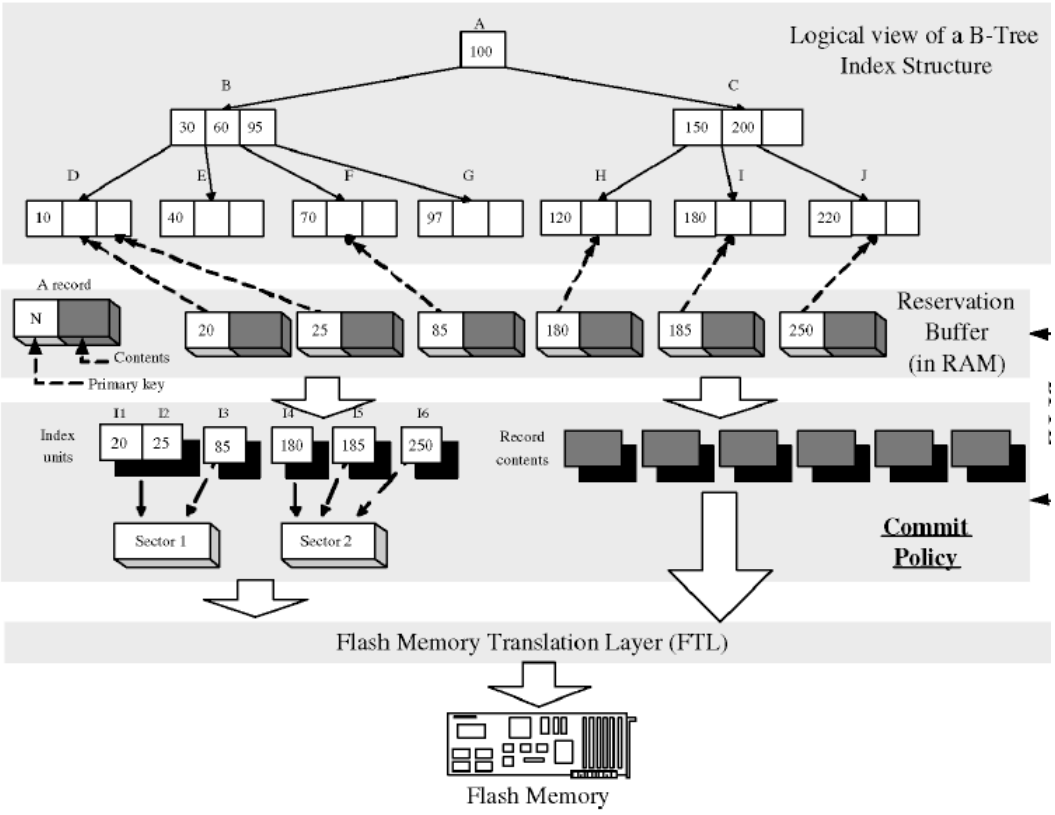



Fig. 3. The commit policy packs and flushes the index units.

### 4.2.1 Commit policy

The reservation buffer is a write buffer residing in main memory.When a B-tree node is inserted, deleted, modified is temporarily held by the reservation buffer. For each record r in reservation buffer, there exits a corresponding B-tree node to which r belongs.The relationship of records in the reservation buffer and B-tree node is maintained for the commit policy.

For each record BFTL will first generate index unit. Based on primary keys of the records and the value range of the leaf node, the index unit will be partitioned in to some disjoint sets. partitioning prevents index units of the same B -tree node from being fragmented. A sector can store fix number of index units. All the index units will be grouped in to few numberof sectors. The capacity of reservation buffer is not unlimited.once it is full, it needs to flush out to the flash memory.

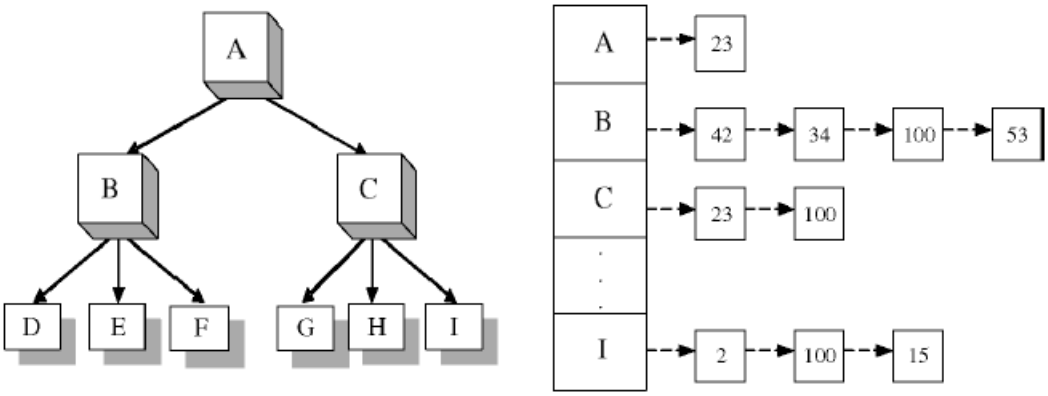
### 4.2.2 The Node Translation Table

A node translation table is adopted to maintain a collection of index units of a B-tree node so that the collecting of index units is efficient.NTT is built in RAM and keeps the list of pages which contain index units belonging to each B-tree node.It maps a B-tree node to a collection of LBA's where the related index units resides.B-tree node consists of several index units, which come from different sectors. The LBA's of the sectors are chained as a list after the corresponding entry of the table. When a B-tree node is visited, all the index unit belonging to the visited node are collected by scanning the sector whose LBA's are stored in the list. In order to form a correct logical view of a B - tree node, BFTL visits all sectors where the related index reside and than construct an up-to-date logical view of the B-tree node. The node translation table is rebuilt when the system is powered up.

**ALGORITHM FOR COMMIT POLICY**

1. Let $\phi$ denote the set of disjoint sets of index units
2. Let $\theta$ denote the set of the sectors
3. Let ntt denote a node translation table
4. while $\phi$ is not empty

   let ds be a disjoint set in $\phi$




(a) Logical view of a B-tree (b) Node translation table

Fig. 4. The node translation table.

let en be the that entry of ntt of a B-tree node that ds would update

**if** length of en is more than C **than** execute compaction

**endif**

**if** there exits a used sector that can store ds **than** ds is stored in

sector sec

record the LBA of sec in the list after the corresponding entry en of ntt

**else**

create a new sector nsec to store ds

record the LBA of nsec in the list after the corresponding entry

en of ntt

$\theta = \theta$+nsec

**endif**

$\phi = \phi$ - ds

**end while**

5. flush out $\theta$ to flash memory

## 4.3 FD-Tree Index Structure

An FD-Tree consists of multiple levels as $L_0 \sim L_{l-1}$.The top level, $L_0$ is a B+ tree called head tree. The size of node in head tree is equal to the size of page in flash disk. Each of the other levels, $L_i$ $(1 \leq i \leq l)$, is a sorted run stored in contiguous pages.Each non-leaf level in FD -tree will have fences (pointer) to point to the next level.

" Each level of FD-tree has a capacity in terms of entries, denoted as $|L_i|$.Following the logrithmic method, we set the levels with a stepped capacity, i.e $|L_{i+1}|$=k $\times$ $|L_i|$ $(0 \leq i \leq l-2)$,where k is the logarithmic size ratio between adjacent levels.Therefore $|L_i| = k^i \times |L_0|$."[yina09].Initially all the updates are performed in the head tree when head tree reaches its maximum capacity entries are migrated from head tree to the sorted run using merging operation where entries are stored in sorted order. Following the design principle P2, the maximum size of the head tree is set to the maximum size of local area like ram memory so that random writes performance is similar to the sequential one.

We categorize the entry in FD- tree in to two kinds,index entry and fence.

- Index Entry. An index entry contains three field: an index key, key, and a record ID, rid, for the indexed data record, and type indicating its role in the logarithmic deletion of FD - tree.Depending on the type, we further categorize index entries into two kinds,filter entries and normal entries.
  - Filter Entry. ($type = Filter$ ) . A Filter entry is a mark of deletion.The filter entry is insertd inn FD-tree upon a deletion to indicate that its corresponding record and index entry have been nominally deleted. It has the same key and the record ID as that deleted index entry.We call that index entry as a phantom entry, as it has been logically deleted but has not been physically removed from the index.
  - Normal Entry. ($type = Normal$ ) All index entries other than filter entries are called normal entries.
- Fence. A fence is an entry with three fields. a key value, key, a type, and a pid, the id of the page in the immediate lower level that a search will go next.Essentially a fence is a pointer, whose key is selected from the index entry in FD-tree.

INVARIANT 1. The first entry of each page is a fence. INVARIANT 2. The key range between a fence and its next fence at the same level is contained in the key range of the page pointed by the fence. Depending on whether the key value of the fence in $L_i$ is selected from $L_i$ or $L_{i+1}$. We categorize fences in $L_i$ into two kinds, internal fence and external fence.

- External Fence. ($type = External$). The key value of an external fence in $L_i$ is selected from $L_{i+1}$. We create a fence for each page of $L_{i+1}$ . For page Pin $L_{i+1}$, we select the key of the first entry in P to be the key of the fence, and set the pid field ofthe fence to be the id of P, in order to satisfy INVARIANT 2.

- Internal Fence. ($type = Internal$). The key value of an internal fence in $L_i$ is selected from $L_i$. If the first entry of any page P is not a external fence, we add an internal fence to the first slot of this page in order to satisfy Invariant 1. The key value of the internal fence is set to be the key of the first index entry e in page P. The pid field of the internal fence is set to the id of the page in the next level whose key range covers the key of e.

### 4.3.1 Operations on FD tree

- Search. For searching an entry in FD-tree first look up opeation is performed in the head tree. Since head tree is B+ tree, searching in head tree is similar to the searching in B+ tree. Than from leaf node of the B+ tree we find best suitable external fence to direct the search to the next level. At each level only one page is accessible if no duplicate is allowed. Since entries are stored in sorted order in pages in sorted runs binary search can be performed to find the greatest key equal to or less than searching element within the page.Than we will scan the page from right to left untill we get a fence entry.Than we will go to the next level guided by the fence entry. Searching an element is performed from top to bottom. We can get filter entry and corresponding phantom entry of same key value and same rid that need to be deleted.

- Insertion. Initially we insert new entry in to the head tree similar to B+ tree.When head tree reaches its maximum capacity we merge head tree with the sorted runs and migrate all the entries from head tree to the sorted run.If first sorted run $L_i$ reaches its maximum capacity merge is performed between $L_{i+1}$.If leaf level exceeds new level will be created. All the entries from head tree is sequentially written to the sorted runs and random writes are limited to the headtree following design principle P2.

- Merge.When a level exceed its capacity it is merged with the next level. The merge operation sequentially scans the two inputs, and combines them into one sorted run in contiguous pages. A newly generated level $L_i$ consists of all index entries from $L_{-1}$, all index entries and external fences from$L_i$. We keep all external fences in $L_i$ because the level ($L_{i+1}$) pointed by these external fences does not change. According to the INVARIANT -1 first entry of the page should be fence entry during merge process first entry of the page will become internal fence if external fence is not possible. This merging can change external fences of upper level up to the root level. That is, given two adjacent levels, $L_{i-1}$ and $L_i$, the merge process generates i + 1 new sorted runs to update all levels from $L_0$ to $L_i$. If the new $L_i$ exceeds its capacity, $L_i$ and $L_{i+1}$ are merged. This process continues until the larger one of the two newly generated levels does not exceed the capacity.

  The merge operation involves only sequential reads and writes, thus we successfully transform the random writes of insertion into sequential reads and writes, following the design principle P1 . We further optimize the I/O performance by applying the multi-page I/O optimization, following our design principle P3 . Since the pages in each level of FD-tree are stored contiguously on the flash disk, we fetch multiple pages in a single I/O request. Similarly, as the newly generated sorted runs are sequentially written, we write multiple pages in a single request. The suitable number of pages in an I/O request is set to be the access unit size when the transfer rate of the sequential access pattern reaches the maximum.

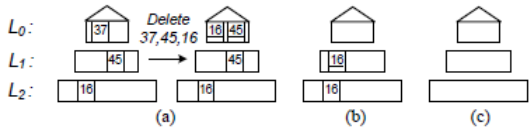



**Figure 3: An example of the logarithmic deletion proces**

- Deletion. For deleting an entry from FD-tree first we insert that entry with same key value and record id to the head tree. and make that entry as a filter entry and migrated to the lower levels as the merge process occurs, and we perform deletion to the lower level. The entry to be deleted is became phantom entry, and is left untouched. Than we perform search on FD-Tree for entry that has to be deleted. The actual deletion is performed in the merge operation when the filter entry encounters its corresponding phantom entry.When filter entry encounters its phantom entry both entries are discarded and will not appear in the merge result. After discarding these entries newly generated sorted run will be sorter than the old one. The space overhead of filter and phantom entry is very low.

**ALGORITHM FOR INSERTION IN FD-TREE**

**Insert(e)**

parameter: e is entry to be inserted

1: Insert e into $L_0$

2: **if** $L_0$ reaches level capacity **then**

Merge($L_0$,$L_1$)

**ALGORITHM FOR MERGING TWO LEVELS**

**Merge($L_{i-1}$,$L_i$)**

Parameter:$L_{i-1}$,$L_i$: the levels to be merged set : x=0 and y=0

1:Suppose $e_{i-1}$[x] and $e_i$[y] is the first element of $L_{i-1}$ and $L_i$

2:Create an array $L_i$'of size $L_i$

3:Repeat the process untill all the elements of $L_{i-1}$ and $L_i$ are visited

4: **while** $e_{i-1}$[x].type=Fence

5: x=x+1

6: **while** $e_i$[y].type=Internalfence

7: y=y+1

8: **if** $e_{i-1}$[x].type=filter and $e_i$[y].type=normal and $e_{i-1}$[x].key=$e_i$[y].key and $e_{i-1}$[x].rid=$e_i$[y].rid **than** entry should be deleted x=x+1 and y=y+1

9: **if** $e_{i-1}$[x].key≤$e_i$[y].key

10: insert_entry=$e_{i-1}$.key

11: x=x+1

12: **else**

13: insert_entry=$e_i$[y].key

14: y=y+1

15: if insert_entry=Fence

16: temp=insert_entry

17: **if** current page in $L_i$'is empty **then**

18: **if** insert_entry ≠ fence and $L_i$[y]≠ leaf

19: internalfence.key=insertentry.key

20: internalfence.rid=insertentry.rid

21: insert internalfence in to $L_i$'

22: insert insert_entry into $L_i$'

23: externalfence.key=insert_entry.key

24: externalfence.rid=ID of the current page in $L_i$'

25: insert externalfence in to $L_{i-1}$

26: update external fence upto the root level

27: **else**

28: insert insert_entry to the $L_i$'

29:**if** $L_i$' reaches its level capacity **then**

30: Merge($L_i$',$L_{i+1}$)

31:Replace $L_i$ by $L_i$

# Chapter 5

# Data Placement Schemes

## 5.1 Data Placement Schemes Used In DBMS

There are two types of data placement schemes used in DBMS.

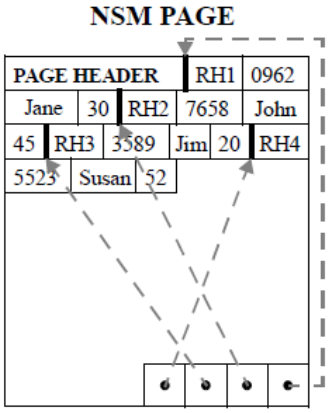
- Row Oriented DBMS
- Column Oriented DBMS

In Row Oriented DBMS a whole row of data has to be read from disk, even if few attributes are required to answer the query. It means system has to read unnecessary attributes to answer the query. It decreases the cache utilization of system.

In Column Oriented DBMS only the required columns are read from the disk. It increases cache utilization of the system. It needs some extra processing to construct the output tuples from these individual columns.

## 5.2 Data Page LayOuts for Relational Database

- N - Ary Storage Model
- Data Decomposition Storage Model
- PaxPage Data Layout




### 5.2.1 N - Ary Storage Model

It is a row oriented storage model also known as Slotted Page storage model. It stores records sequentially in disk. Each record has a record header that contains a null bitmap, offset to the variable length value, and other implementation specific information. Each record is inserted when available free space is greater than length of the record. Offset of the begning of record is inserted into next available slot from the end of the page. The nth record of the page will be accessed by the nth pointer from the end of the page. During the predicate evaluation it exibhit poor cache performance.

consider a query.

**select name from R where age $\leq$ 40.**

To evaluate this query query processor uses a scan operator, that evaluates predicate and retrieves age attribute. Suppose NSM page is already in memory and record size is smaller than cache block size.Suppose age attribute size is 4 - byte and cache block size is 16 - byte. When scan operator retrieves age attribute it will bring other attributes stored next to age. It will waste cache space to store non-reference data, and incurring unnecessary access to main memory.

### 5.2.2 The Decomposition Storage Model

It uses vertical partitioning of relation. It partitions relation into n sub-relations where n is the number of attributes in that relation. Vertical Partitioning was proposed in order to reduce I/O cost and to improve cache performance of the system. DSM offers a heigher degree of spatial locality when sequential accessing

the values of attribute. DSM performance is superior to NSM when queries access is fewer than 10% of the attributes in each relation. When evaluating multi attribute query it gives poor performance due to reconstruction cost of each subrelations. The time spent on joining increases with the number of attributes increases in the result relation. In addition, DSM incurs a significant space overhead because the record id of each record needs to be replicated.

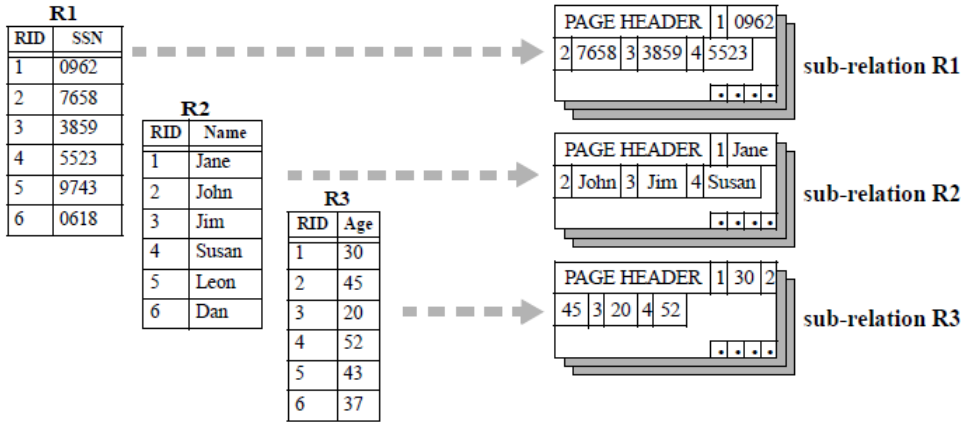



**FIGURE 2: The Decomposition Storage Model (DSM).** *The relation is partitioned vertically into one thin relation per attribute. Each sub-relation is then stored in the traditional fashion.*

### 5.2.3 PaxPage Storage Model:-

Following are the main advantages of PaxPage layout.

- Maximize inter record spatial locality within each column and within each page which eliminates unnecessary request to main memory.
- Minimizes record reconstruction cost.
- It is orthogonal to design decision, because it only affects the layout of data stored on the single page.

The motivation behind PAX is to keep the all attributes of the relation into single page. It partitions the page vertiacally into n minipages where n is the number of attributes in that relation. In PAX each records resides on the same page as it would resides when NSM were used. Pax improves cache performance. Pax inceases inter-record spatial locality. It employs in-page vertical partitioning, which reduces reconstruction cost.

**Design Of Pax Page :-**In order to store a relation with n attributes, Pax partitions each page in to n minipages. Values of the first attribute will be in first minipage and so on. At the begining of the each page there is a page header that contains offset to the beginning of each minipages. The

record header information is distributed across the minipages. The structure of minipages will be defined as follows:-

There are two types of attribbutes:-

- Fixed Length Attribute
- Variable Length Attribute

Fixed Length Attributes are stored in F-minipages. At the end of each minipage there is a presence bit vector with one entry per record that show null value if that attibute value is not present.

Variable length attributes are stored in V-minipages. V - minipages are slotted. Offset of each value of a v-minipage is stored at the end of that v-minipage. Null vaues are denoted by null pointer.
Each page contains a page header and number of minipages equal to the number of attributes of relation. The page header contains the number of attributes, attribute size (for fixed length attribute), offset to the begining of the minipages, the total number of records on the page and the total space still available.
Given figure depicts an example where two records have been inserted, where two attributes are of fixed length size and one attribute is variable length size. There is two f - minipages and one v - minipage. At the end of v - minipage there are offsets to the end of each variable length value. Records on a page are accessed sequentially or in a random order. In sequentially access a subsets of attributes, the algorithm sequentially access values in the appropriate minipages.
To store relation space required for PAX and NSM are same. NSM stores records continuosly so it requires one offset per record and one additional offset for variable length attribute in each record. PAX on the other hand stores one offset for each variable length attribute and one offset for each of the n - minipages. So NSM and PAX will occupy same amount of space for storing relation. Size of minipages will be calculated on average sizes of attribute values.

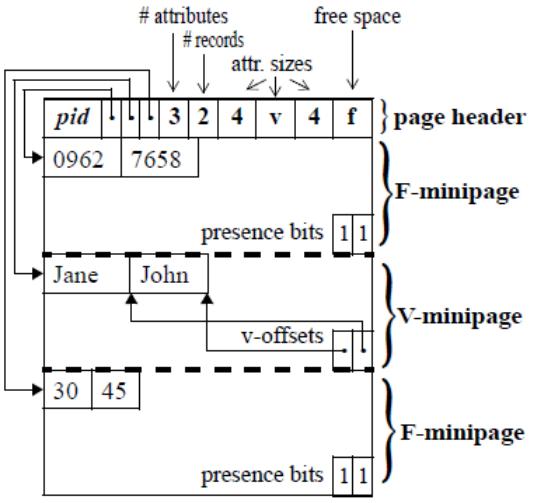



**FIGURE 4: An example PAX page.** *In addition to the traditional information (page id, number of records in the page, attribute sizes, free space, etc.) the page header maintains offsets to the beginning of each minipage.*

# Chapter 6

# Flash Aware Query Processing

Our investigation is on query processing techniques that improve the performance of complex data analysis of dataware houses and buiseness intellegence workloads. Towards this goal, we will evaluate data structures and algorithms that gives fast random read to speed up selection, projection, and join operations.Sorts and aggregation are the most important in complex query plan. We are considering "SSD only DBMS" where all the datas are stored in SSD. We wil investigate which query processing technique gives better performance in SSD. We have discussed PAX PAGE layout which mprove cache performance of the system. In SSD, PAX PAGE layout will reduce amount of data read from SSDs.

## 6.1 Scan Selections And Projections

There is flash scan operator that is used in PAX PAGE layout to improve selection and projection on flash SSD.

### 6.1.1 FlashScan Operator

In column based page layout PAX page is devided in to minipages where values of each attribute are physically separeted within the same page, thus it allows the operator to acccess only the attributes that are needed to answer the query, which reduces memory bandwidth requirement. Data transfer unit in main memory can be as small as 32 to 128 bytes, whereas a database page is typically 8K to 128K .With SSDs, the minimum transefer unit is 512 to 2K bytes, which allows to selectively

read only parts of the regular page.
FlashScan takes advantages of the small transfer unit of SSds to read only the minipages of the attributes that it needs. Suppose there is scan without select predicate that projects first and third attribute of the relation. For each scan, flashscan read minipage for the first attribute and than seeks to the start of the third minipage. similarly it reads for all the pages of that relation.

### 6.1.2 FlashScan And Column Store :-

FlashScan's behaviour in SSD is in many case similar to a column store scan on traditional disk. FlashScan needs to seek between minipages, and that seek is very fast, adds a small overheads but it is preferable over a full sequential access. Column-stores on HDDs read a large portion of a single column at a time to amortize the read disk head seek between different columns. These two scans will perform similarly for both SSDs and HDDs with comparable bandwidths. In heighly selective scans Flash Scan can skip some minipages , that do not contain the qualifying tuple. A column-store on HDDs can only skip entire chunks which are two orders of magnitude larger than flash SSd transfer unit. It means that column store scans on HDDs need to be 100 times more selective than FlashScan to witness similar benefits.

## 6.2 FlashJoin

FlashJoin is a multiway join algorithm for solid state drives. The component of the flash join is decribed below.

### 6.2.1 FlashJoin Overview

Flash Join takes advantages of the fast random read of the SSD. It uses same method like flashscan and avoid reading unnecessary attributes from the relation and post-pones retriving tuples attributes as long as possible. FlashJoin is the multi way equi join, implemented as a pipeline of the stylized binary joins. Each binary join is broken in to two parts.

1. Join Kernel

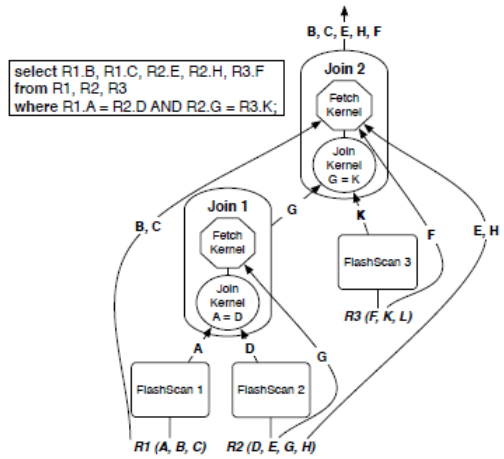



**Figure 6: Execution strategy for a three-way join when FlashJoin is employed.**

2. Materialization Strategy

3. Fetch Kernel

### 6.2.2 JoinKernel

Join Kernel stores only page id and slot no of the qualifying tuples. Group of pageid and slotnumber is called join index. This join can be implemted from any existing join algorithm like block nested loop, sort merge, hybrid hash etc. In our implemtation we have nested loop join to produce join indexes for join kernel.

### 6.2.3 Materialization Strategy

In FlashJoin we use late materialization strategy. This heuristics postpones retrieving attributes as down stream as possible. In late materialization each join passes only page numbers and rids of the given base relation. Attributes will be retrieved as late as possible.

### 6.2.4 FetchKernel

The fetchkernel uses join index, produced by the join kernel to retrive the attributes from join result. FetchKernel Retrives the attribute values whenever it is needed. For each join index fetch kernel locates the needed minipages in the bufferpool and then compose the resulting tuples. This approach is reasonable when all the pages

needed to generate the tuple can be fit in to the main memory because random read are cheap.

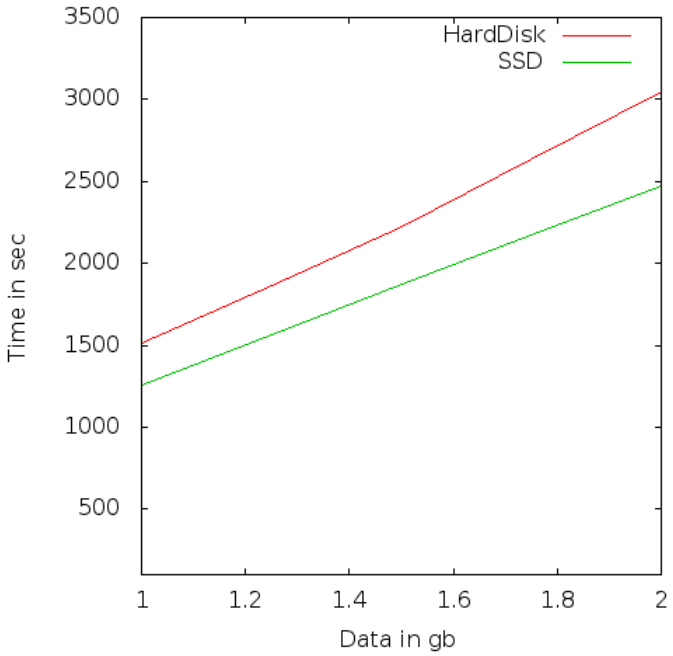
This approache offers some benefits over traiditional joins since we are accessing only needed attribute through rid in flashjoin. For storing records we are using PaxPage layout that will also improve cache utilization of the system.

# Chapter 7

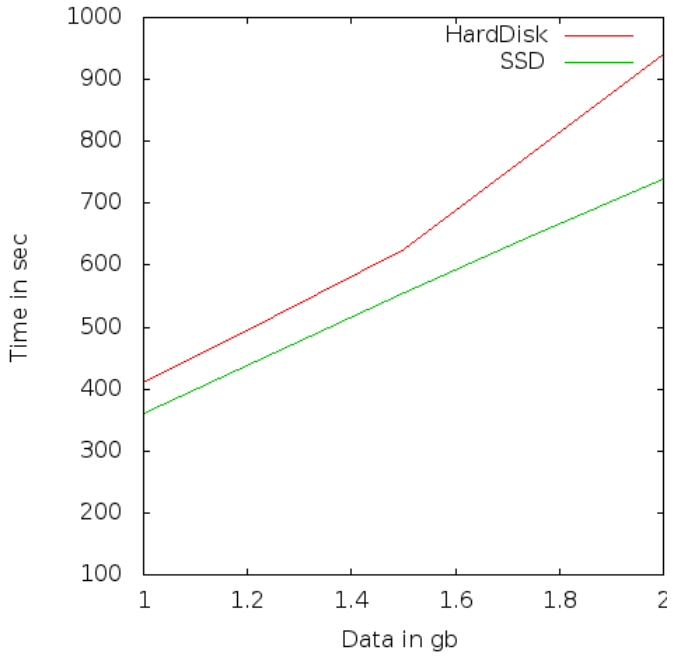
# Implementation And Experimental Results

## 7.1 Read Write Performance Of SSD and Hard-Disk

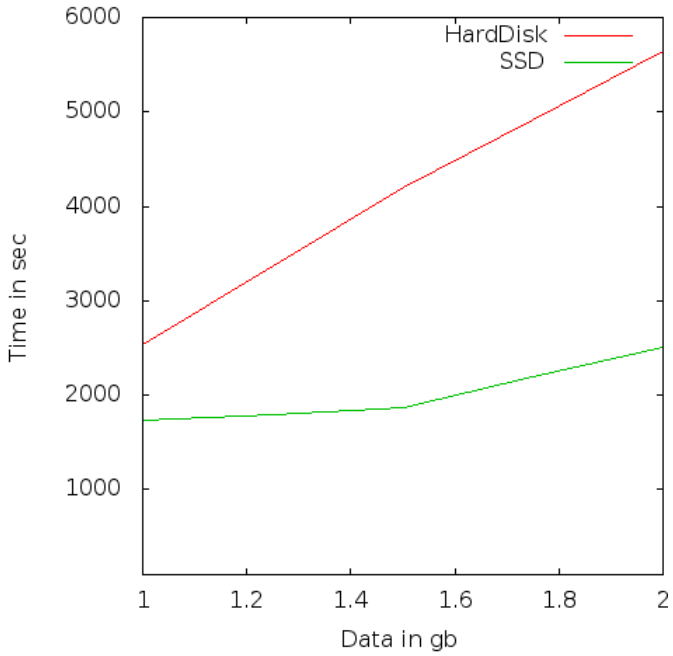
Some experimental results of read write performance of SSD and hard disk are given below Randomwrite performance of SSD and HardDisk is given in graph.



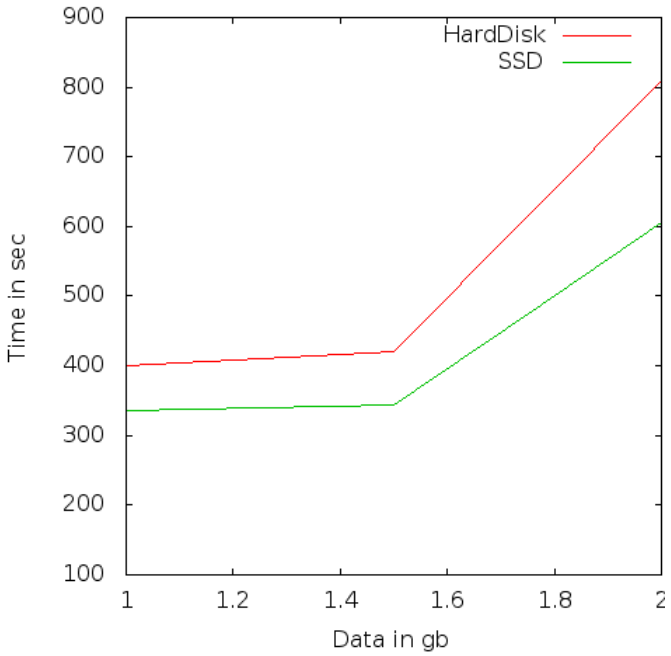

Randomread performance of SSD and HardDisk is given in graph.




Sequential Write performance comparision between SSD and Hard Disk.




Sequential Read performance comparision between SSD and Hard Disk.




## 7.2 Implementation Details

### 7.2.1 Mubase Details

1. **Storage Manager Of Mubase** It will keep track of which block is used by which objects.This object can be a table or an index structure. In index structure it will maintain root page id of each index structure. Initially it will link all the pages in link list. It will also maintain free pages available in disk. It will make use of buffer manager to fetch the disk block in memory and write them back.

2. **Buffer Manager Of Mubase** LRU replacement policy has been implemented. Pinpage of buffer manager will bring the page into memory(if that page is not already there.)It will increment the pincount of that page and returns the address of that page. Unpinpage of buffer manager will reduce the pincount of that block, if pincount is zero that buffer block will be available for eviction.

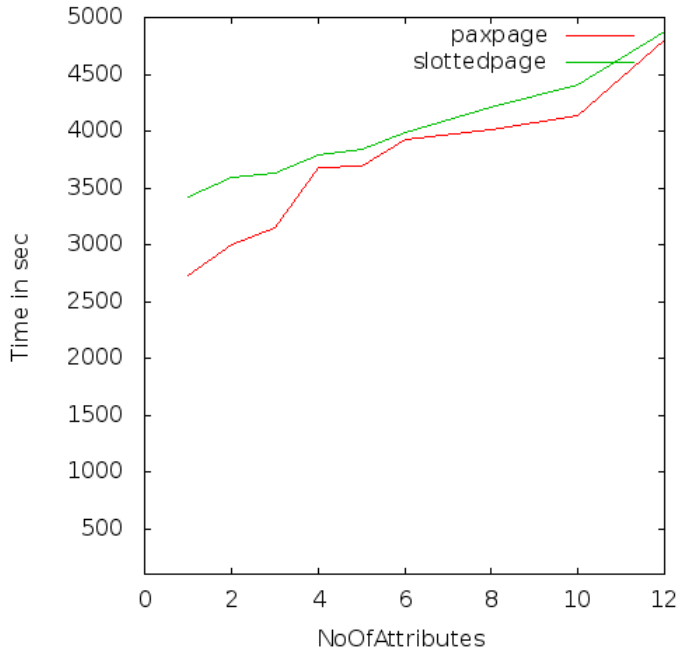
3. **Disk Manager Of Mubase** It will direct write disk blocks to the file and direct read from the file.

4. **Relation Manager Of Mubase** It will create schema for the particular

table to store in the database. All the entries of the tables will be stored and retrieved using relation manager. Joining of the tables has also been implemented in relation manager.

### 7.2.2 Pax Page And Slotted Page Experimental Results

PaxPage and Slotted page have been implemted for storing TPC-H lineitem table into the disk. These are the pag layouts used to store rows of the table into the disk. They use all the layers of mubase to store records. We performed queries for retrieving single attribute and two attributes and so on. We tasted paxpage on solid state drive and slottedpage on harddisk. Given graph will show performance comparision between paxpage and slotted page. The experimental results show total elapse time for retrieving records from the paxpage is less than the slotted page.




### 7.2.3 FlashJoin for PaxPage and NormalJoin for SlottedPage

We implemented flashjoin for paxpage and normal join for slotted page. Experimental results shows that flashjoin on paxpage is 20 % faster than normal join in slotted page.

## 7.3 Indexing Structure

Indices are used to retrieve data from the large database. B+ tree is a indexing structure for magnetic disk .If we use B+ tree index structure for flash disk it will benefit from fast random read speed for search performance but will suffer form poor random write speed in update performance. To optimized the update performance while preserving search performance FD - tree will be suitable for flash disk. We implemented FD - tree index structure.

# Bibliography

[1] [Agra 08] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy. Design tradeoffs for SSD performance. In: ATC08: USENIX 2008 Annual Technical Conference on Annual Technical Conference, pp. 5770, USENIX Association, Berkeley, CA, USA, 2008.

[2] [Boug 09] L. Bouganim, B. r Jnsson, and P. Bonnet. uFLIP: Understanding Flash IO Patterns.. In: CIDR, www.crdrdb.org, 2009.

[3] [Deve 09] R. S. Y. D. Devesh Agrawal, Deepak Ganesan. Lazy-Adaptive Tree: An Optimized Index Structure for Flash Devices. 2009. Proceedings of the 35th International Conference on Very Large Databases (VLDB), Lyon, France.

[4] [Grae 04] G. Graefe. Write-optimized B-trees. In: VLDB 04: Proceedings of the Thirtieth interna- tional conference on Very large data bases, pp. 672683, VLDB Endowment, 2004.

[5] [Grae 07] G. Graefe. The five-minute rule twenty years later, and how flash memory changes the rules. In: DaMoN 07: Proceedings of the 3rd international workshop on Data management on new hardware, pp. 19, ACM, New York, NY, USA, 2007.

[6] [Gray 08] J. Gray and B. Fitzgerald. Flash Disk Opportunity for Server Applications. Queue, Vol. 6, No. 4, pp. 1823, 2008.

[7] [Inte 98] Intel-Corporation. Understanding the Flash Translation Layer(FTL) specifications. 1998. www.embeddedfreebsd.org/Documents/Intel-FTL.pdf.

[8] [Jaga 97] H. V. Jagadish, P. P. S. Narayan, S. Seshadri, S. Sudarshan, and R. Kanneganti. Incremental Organization for Data Recording and Warehousing. In: VLDB 97: Proceedings of the 23rd International Conference on Very Large

Data Bases, pp. 1625, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[9] [Lee 07] S.-W. Lee and B. Moon. Design of flash-based DBMS: an in-page logging approach. In: SIG- MOD 07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data, pp. 5566, ACM, New York, NY, USA, 2007.

[10] [Lee 08] S.-W. Lee, B. Moon, C. Park, J.-M. Kim, and S.-W. Kim. A case for flash memory ssd in enterprise database applications. In: SIGMOD 08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data, pp. 10751086, ACM, New York, NY, USA, 2008.

[11] [Leve 08] A. Leventhal. Flash storage memory. Commun. ACM, Vol. 51, No. 7, pp. 4751, 2008.

[12] [Mosh 08] M. Moshayedi and P. Wilkison. Enterprise SSDs. Queue, Vol. 6, No. 4, pp. 3239, 2008.

[13] [Myer 07] D. Myers. On the Use of NAND Flash Memory in High-Performance Relational Databases: MSc Thesis. 2007. http://people.csail.mit.edu/dsm/flash-thesis.pdf.

[14] [Nath 07] S. Nath and A. Kansal. FlashDB: dynamic self-tuning database for NAND flash. In: IPSN 07: Proceedings of the 6th international conference on Information processing in sensor networks, pp. 410419, ACM, New York, NY, USA, 2007.

[15] [Norh 08] H. O. Norheim. How Flash Memory Changes the DBMS 1 World. 2008. www.hansolav.net/blog/content/binary/HowFlashMemory.pdf

[16] [ONei 96] P. ONeil, E. Cheng, D. Gawlick, and E. ONeil. The log-structured merge-tree (LSM-tree). Acta Inf., Vol. 33, No. 4, pp. 351385, 1996.

[17] [Shah 08] M. A. Shah, S. Harizopoulos, J. L. Wiener, and G. Graefe. Fast scans and joins using flash drives. In: DaMoN 08: Proceedings of the 4th international workshop on Data management on new hardware, pp. 1724, ACM, New York, NY, USA, 2008.

[18] [Silb 05] A. Silberschatz, H. F. Korth, and S. Sudarshan. Database System Concepts. McGraw-Hill, Inc., New York, NY, USA, 2005.

[19] [Wu 07] C.-H. Wu, T.-W. Kuo, and L. P. Chang. An efficient B-tree layer implementation for flash- memory storage systems. Trans. on Embedded Computing Sys., Vol. 6, No. 3, p. 19, 2007.

[20] [Yin 08] S. Yin, P. Pucheral, and X. Meng. PBFilter: indexing flash-resident data through partitioned summaries. In: CIKM 08: Proceeding of the 17th ACM conference on Information and knowledge management, pp. 13331334, ACM, New York, NY, USA, 2008.

[21] [Yina 09] Q. L. Yinan Li, Bingsheng He and K. Yi. Tree Indexing on Flash Disks. 2009. IEEE International Conference on Data Engineering (ICDE).

[22] Query Processing Techniques For Solid state drive Dimitris Tsirogiannis University of Toronto Toronto, ON, Canada dimitris@cs.toronto.edu Stavros Harizopoulos HP Labs Palo Alto, CA, USA stavros@hp.com Mehul A. Shah HP Labs Palo Alto, CA, USA mehul.shah@hp.com